18 | 异步编程（一）：V8是如何实现微任务的?

2020-04-25 李兵

图解 Google V8

进入课程 >

讲述：李兵
时长 16:26　大小 15.06M

你好，我是李兵。

上节我们介绍了通用的 UI 线程架构，每个 UI 线程都拥有一个消息队列，所有的待执行的事件都会被添加进消息队列中，UI 线程会按照一定规则，循环地取出消息队列中的事件，并执行事件。而 JavaScript 最初也是运行在 UI 线程中的。换句话说，JavaScript 语言就是基于这套通用的 UI 线程架构而设计的。

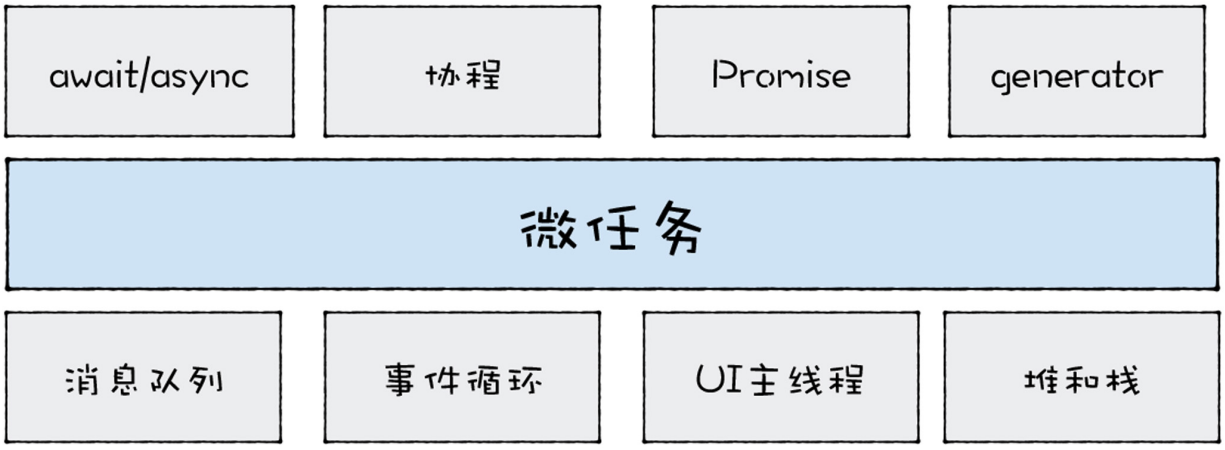
基于这套基础 UI 框架，JavaScript 又延伸出很多新的技术，其中应用最广泛的当属**宏**和**微任务**。

宏任务很简单，**就是指消息队列中的等待被主线程执行的事件。**每个宏任务在执行时，V8都会重新创建栈，然后随着宏任务中函数调用，栈也随之变化，最终，当该宏任务执行结束时，整个栈又会被清空，接着主线程继续执行下一个宏任务。

微任务稍微复杂一点，其实你可以把**微任务看成是一个需要异步执行的函数，执行时机是在主函数执行结束之后、当前宏任务结束之前。**

JavaScript 中之所以要引入微任务，主要是由于主线程执行消息队列中宏任务的时间颗粒度太粗了，无法胜任一些对精度和实时性要求较高的场景，那么**微任务可以在实时性和效率之间做一个有效的权衡**。另外使用微任务，可以改变我们现在的异步编程模型，使得我们可以使用同步形式的代码来编写异步调用。

虽然微任务如此重要，但是理解起来并不是太容易。我们先看下和微任务相关的知识栈，具体内容如下图所示：



从图中可以看出，微任务是基于消息队列、事件循环、UI 主线程还有堆栈而来的，然后基于微任务，又可以延伸出协程、Promise、Generator、await/async 等现代前端经常使用的一些技术。也就是说，如果对**消息队列、主线程**还有**调用栈**理解的不够深入，你在研究微任务时，就容易一头雾水。

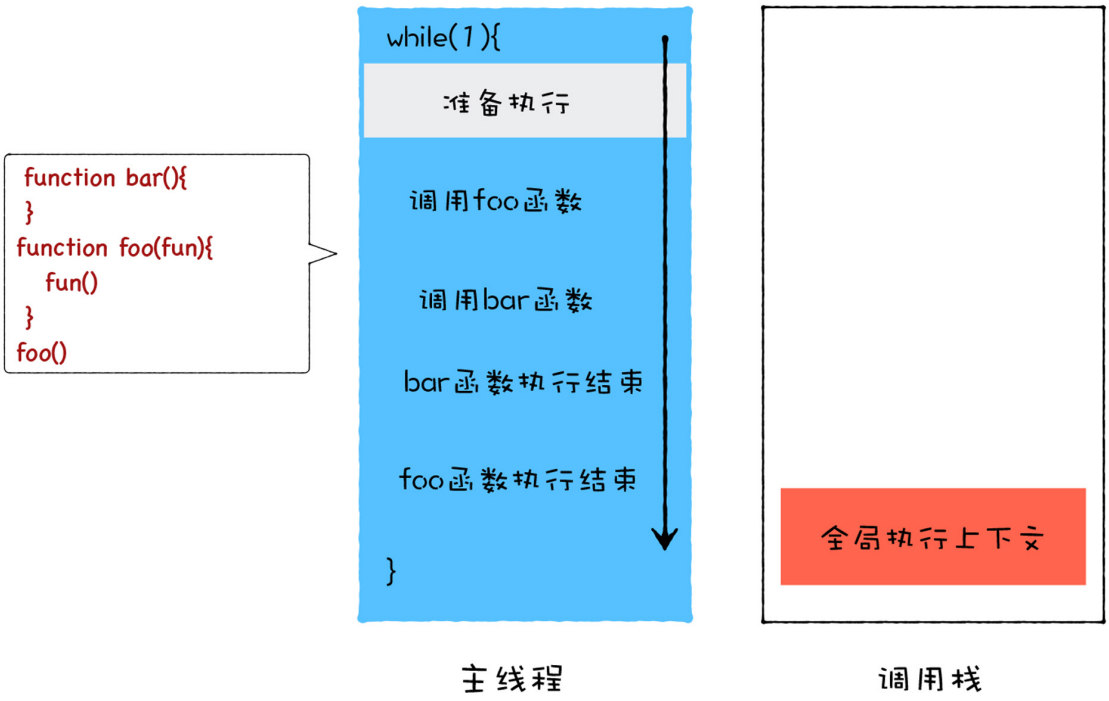
今天，我们就先来打通微任务的底层技术，搞懂消息队列、主线程、调用栈的关联，然后抽丝剥茧地剖析微任务的实现机制。

主线程、调用栈、消息队列

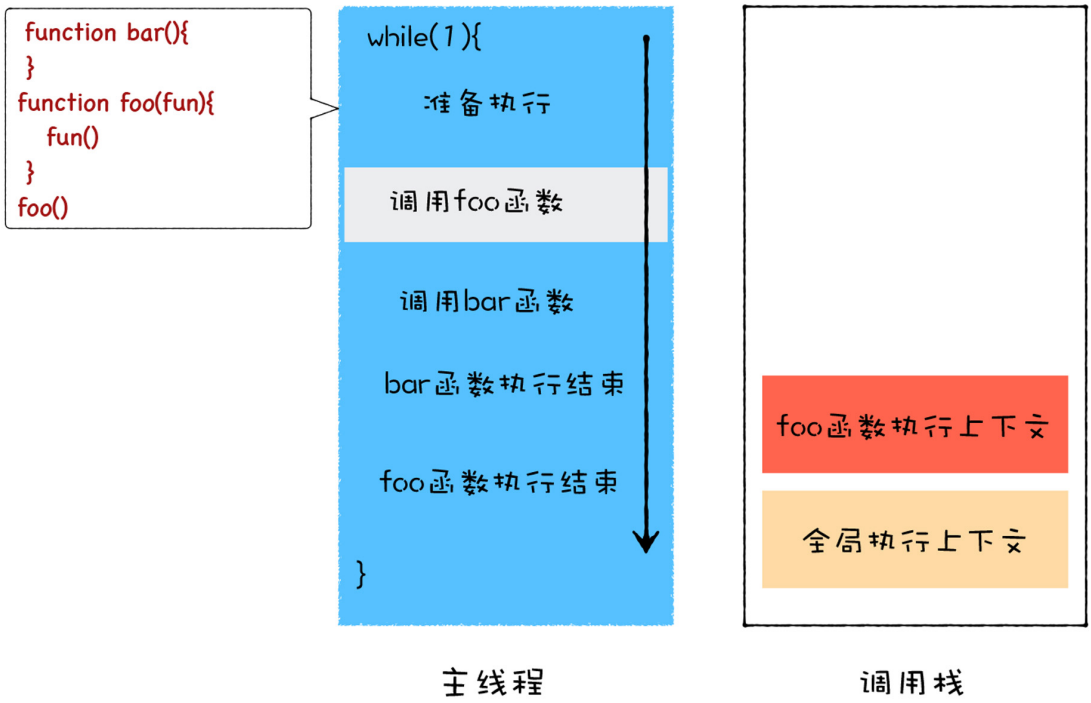
我们先从**主线程**和**调用栈**开始分析。我们知道，**调用栈是一种数据结构，用来管理在主线程上执行的函数的调用关系。**接下来我们通过执行下面这段代码，来分析下调用栈是如何管理主线程上函数调用的。

```
function bar() {
}
foo(fun){
  fun()
}
foo(bar)
```

当 V8 准备执行这段代码时，会先将全局执行上下文压入到调用栈中，如下图所示：

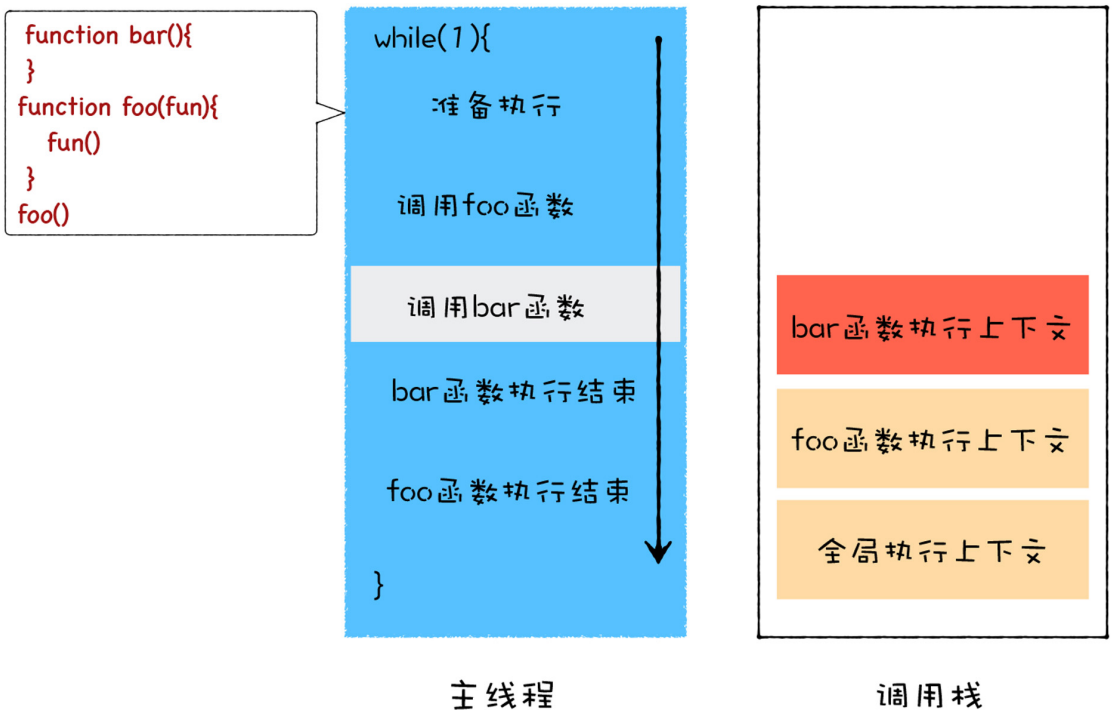


然后 V8 便开始在主线程上执行 foo 函数，首先它会创建 foo 函数的执行上下文，并将其压入栈中，那么此时调用栈、主线程的关系如下图所示：



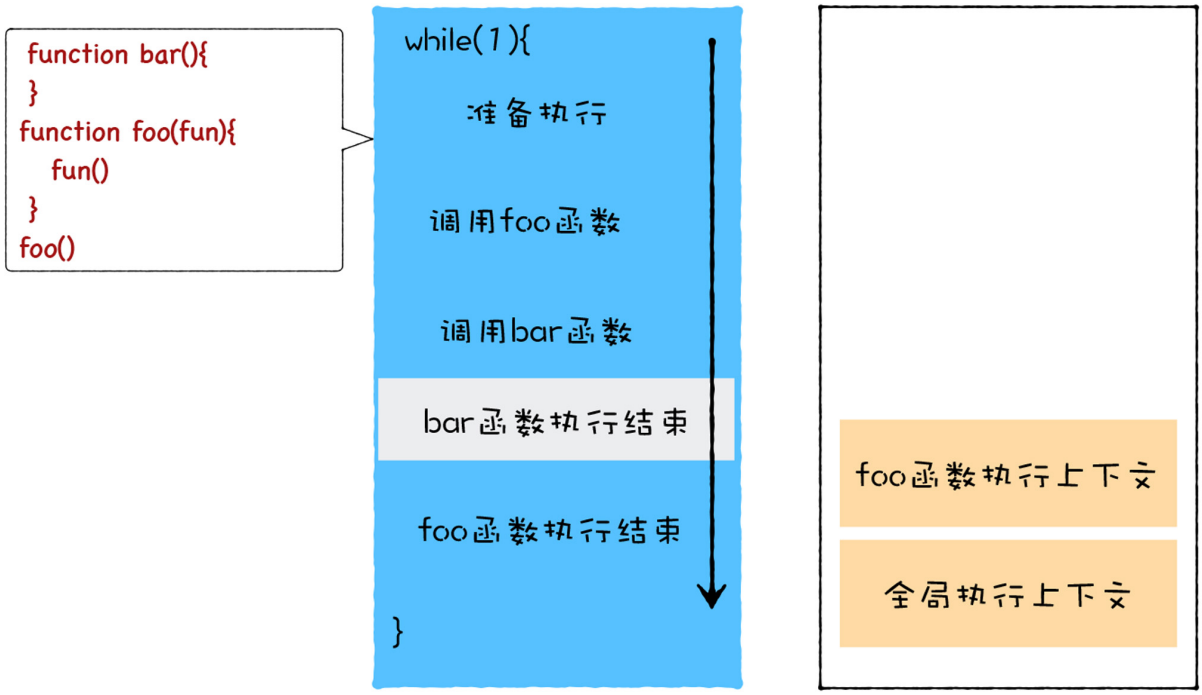
准备执行

然后，foo 函数又调用了 bar 函数，那么当 V8 执行 bar 函数时，同样要创建 bar 函数的执行上下文，并将其压入栈中，最终效果如下图所示：



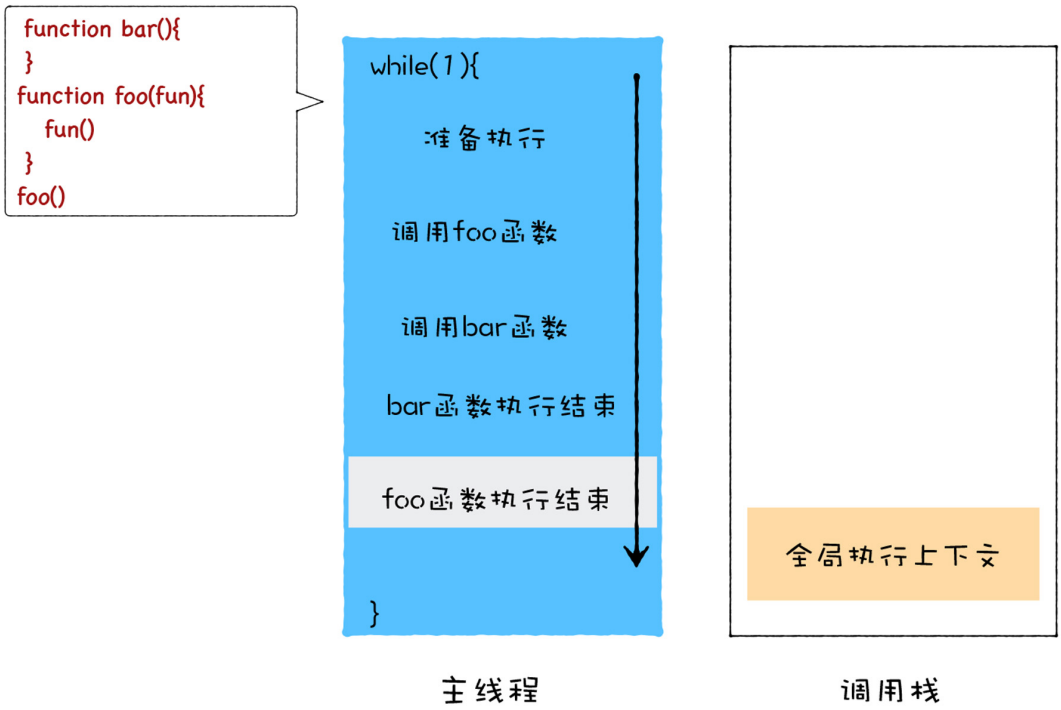
调用bar函数

等 bar 函数执行结束，V8 就会从栈中弹出 bar 函数的执行上下文，此时的效果如下所示：



bar函数执行结束

最后，foo 函数执行结束，V8 会将 foo 函数的执行上下文从栈中弹出，效果如下所示：



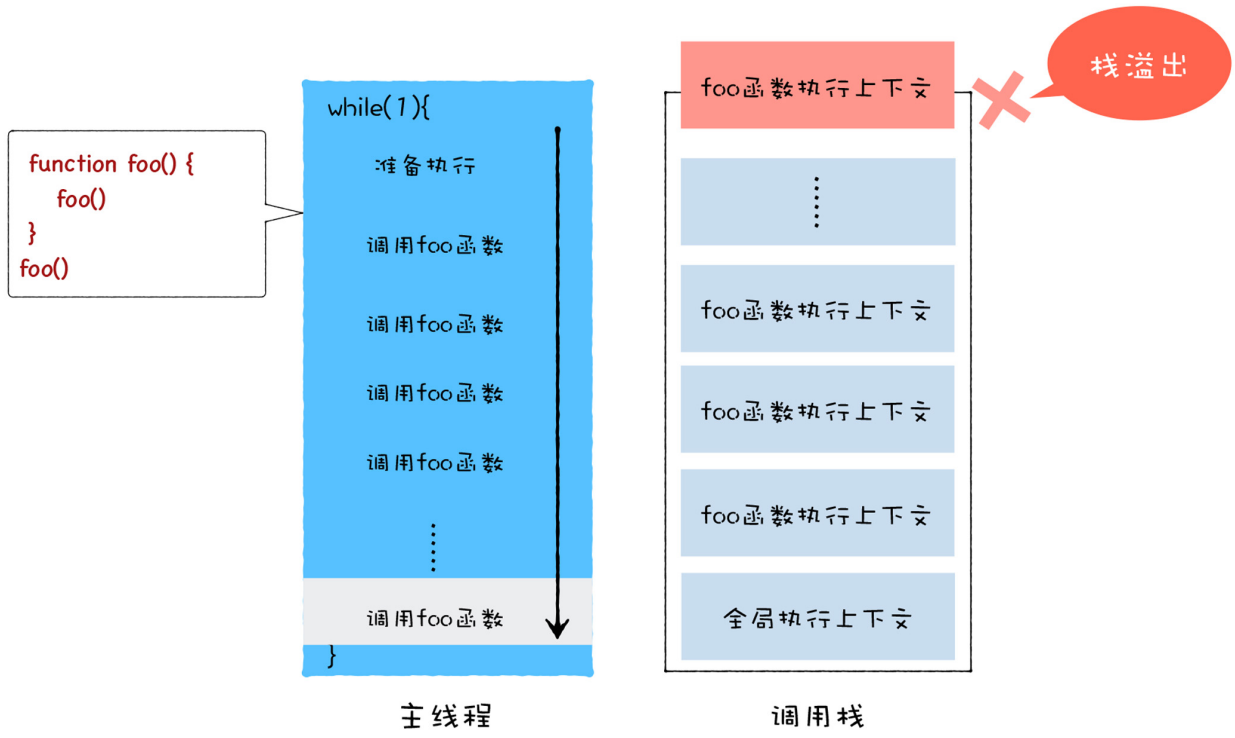
foo函数执行结束

以上就是调用栈管理主线程上函数调用的方式，不过，这种方式会带来一种问题，那就是栈溢出。比如下面这段代码：

```
function foo(){
  foo()
}
foo()
```

复制代码

由于 foo 函数内部嵌套调用它自己，所以在调用 foo 函数的时候，它的栈会一直向上增长，但是由于栈空间在内存中是连续的，所以通常我们都会限制调用栈的大小，如果当函数嵌套层数过深时，过多的执行上下文堆积在栈中便会导致栈溢出，最终如下图所示：



栈溢出

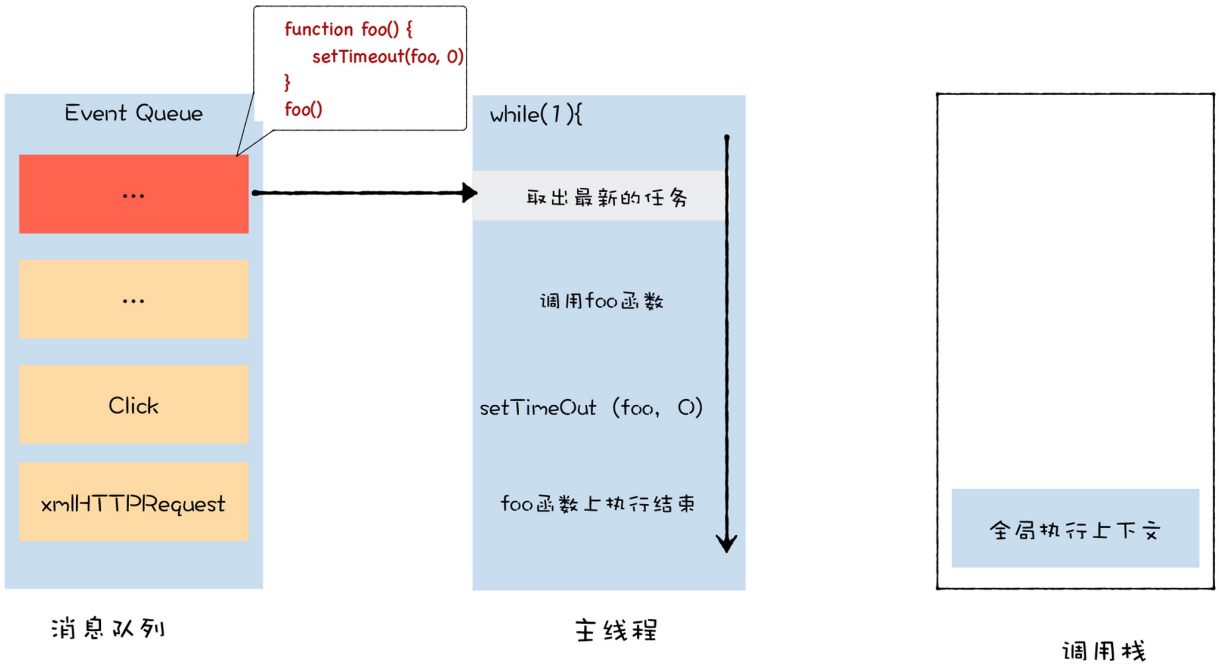
我们可以使用 setTimeout 来解决栈溢出的问题，setTimeout 的本质是将同步函数调用改成异步函数调用，这里的异步调用是将 foo 封装成事件，并将其添加进**消息队列**中，然后主线程再按照一定规则循环地从消息队列中读取下一个任务。使用 setTimeout 改造后代码代码如下所示：

```
function foo() {
  setTimeout(foo, 0)
}
foo()
```

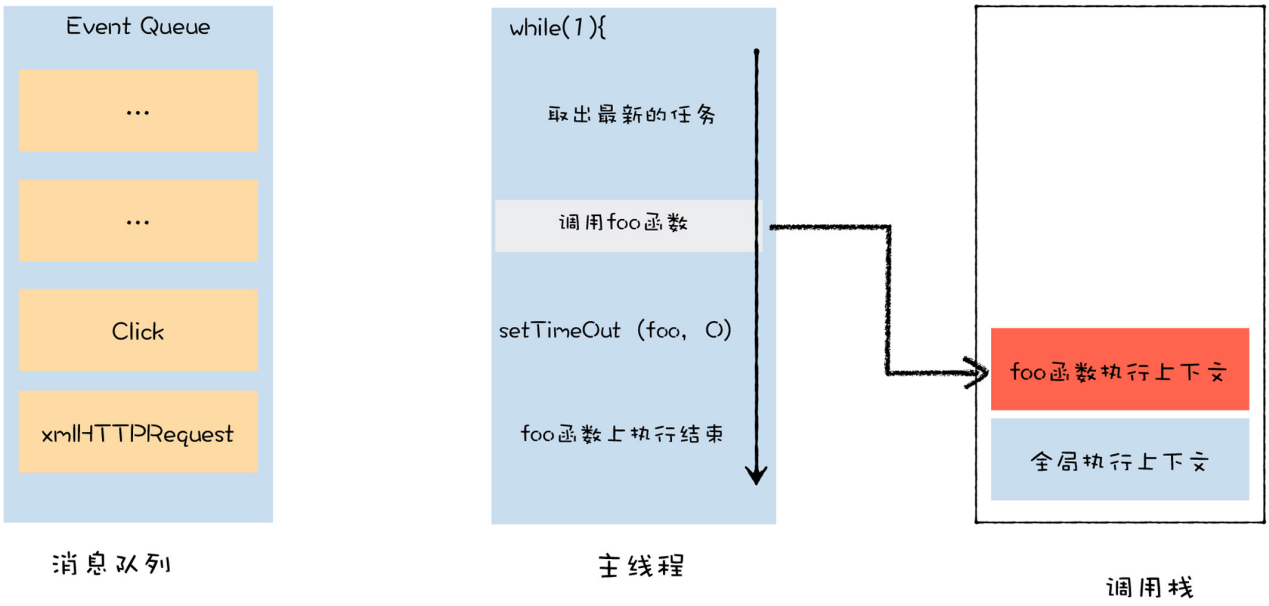
复制代码

那么现在我们就可以从**调用栈**、**主线程**、**消息队列**这三者的角度来分析这段代码的执行流程了。

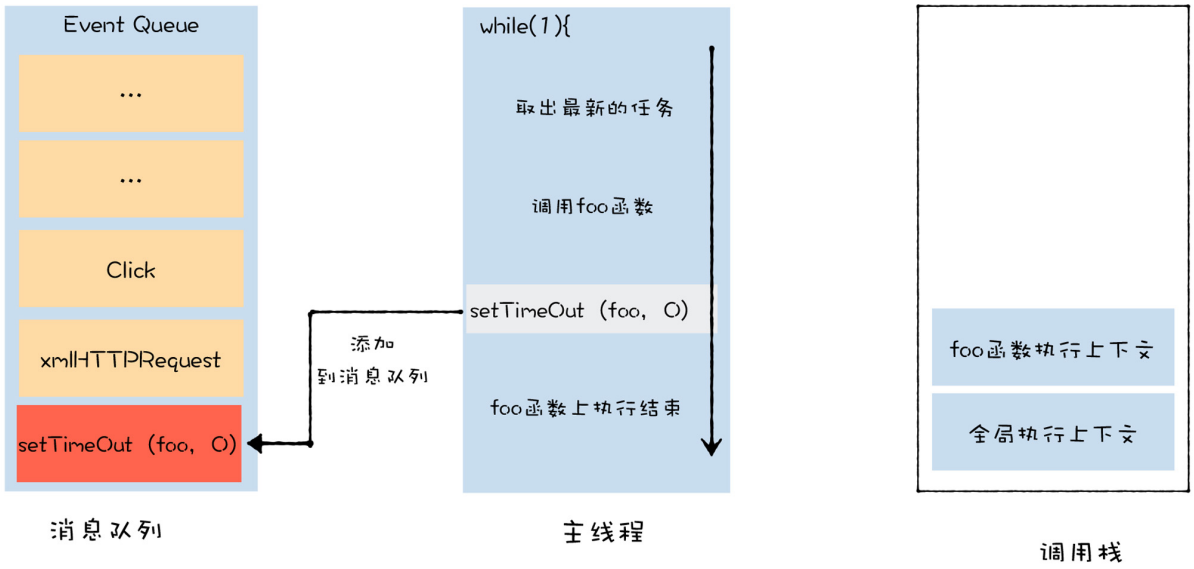
首先，主线程会从消息队列中取出需要执行的宏任务，假设当前取出的任务就是要执行的这段代码，这时候主线程便会进入代码的执行状态。这时关于主线程、消息队列、调用栈的关系如下图所示：



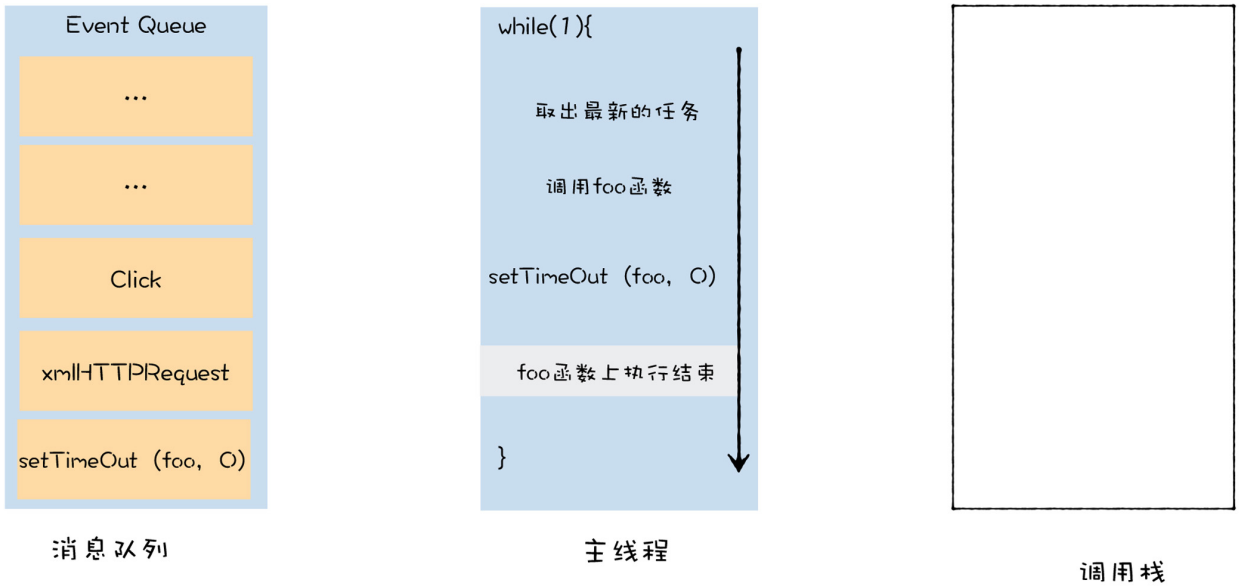
接下来 V8 就要执行 foo 函数了，同样执行 foo 函数时，会创建 foo 函数的执行上下文，并将其压入栈中，最终效果如下图所示：



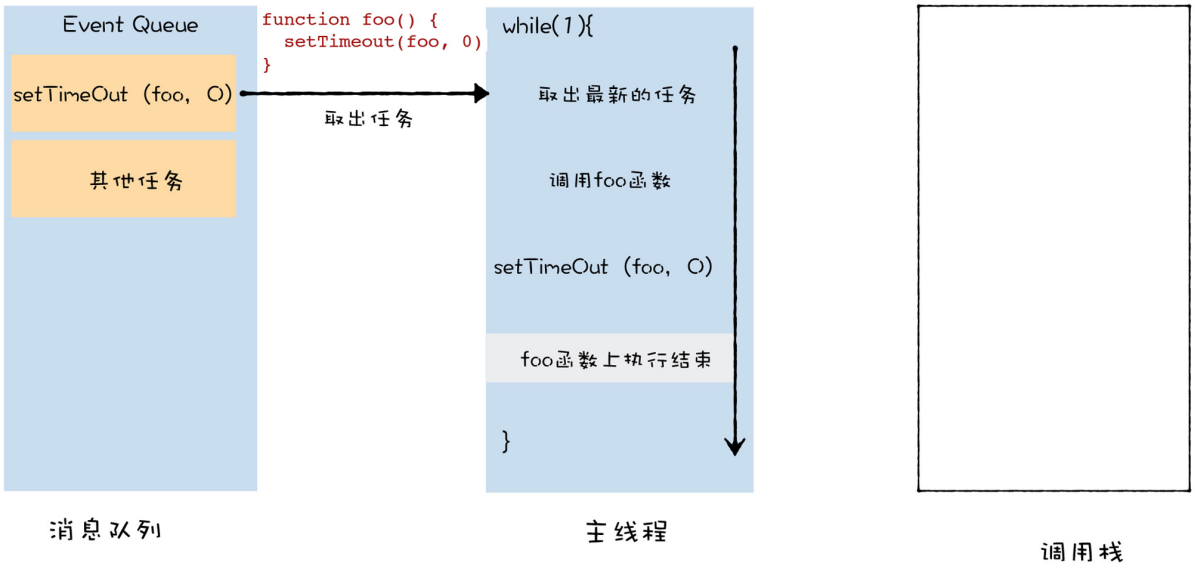
当 V8 执行执行 foo 函数中的 setTimeout 时，setTimeout 会将 foo 函数封装成一个新的宏任务，并将其添加到消息队列中，在 V8 执行 setTimeout 函数时的状态图如下所示：



等 foo 函数执行结束，V8 就会结束当前的宏任务，调用栈也会被清空，调用栈被清空后状态如下图所示：



当一个宏任务执行结束之后，忙碌的主线程依然不会闲下来，它会一直重复这个取宏任务、执行宏任务的过程。刚才通过 setTimeout 封装的回调宏任务，也会在某一时刻被主线取出并执行，这个执行过程，就是 foo 函数的调用过程。具体示意图如下所示：



因为 foo 函数并不是在当前的父函数内部被执行的，而是封装成了宏任务，并丢进了消息队列中，然后等待主线程从消息队列中取出该任务，再执行该回调函数 foo，这样就解决了栈溢出的问题。

微任务解决了宏任务执行时机不可控的问题

不过，对于栈溢出问题，虽然我们可以通过将某些函数封装成宏任务的方式来解决，但是宏任务需要先被放到消息队列中，如果某些宏任务的执行时间过久，那么就会影响到消息队列后面的宏任务的执行，而且这个影响是不可控的，因为你无法知道前面的宏任务需要多久才能执行完成。

于是 JavaScript 中又引入了微任务，微任务会在当前的任务快要执行结束时执行，利用微任务，你就能比较精准地控制你的回调函数的执行时机。

通俗地理解，V8 会为每个宏任务维护一个微任务队列。当 V8 执行一段 JavaScript 时，会为这段代码创建一个环境对象，微任务队列就是存放在该环境对象中的。当你通过 Promise.resolve 生成一个微任务，该微任务会被 V8 自动添加进微任务队列，等整段代码快要执行结束时，该环境对象也随之被销毁，但是在销毁之前，V8 会先处理微任务队列中的微任务。

理解微任务的执行时机，你只需要记住以下两点：

首先，如果当前的任务中产生了一个微任务，通过 Promise.resolve() 或者 Promise.reject() 都会触发微任务，触发的微任务不会在当前的函数中被执行，所以执行微任务时，不会导致栈的无限扩张；

其次，和异步调用不同，微任务依然会在当前任务执行结束之前被执行，这也就意味着在当前微任务执行结束之前，消息队列中的其他任务是不可能被执行的。

因此在函数内部触发的微任务，一定比在函数内部触发的宏任务要优先执行。为了验证这个观点，我们来分析一段代码：

复制代码

```
function bar(){
  console.log('bar')
  Promise.resolve().then(
    (str) =>console.log('micro-bar')
  )
  setTimeout((str) =>console.log('macro-bar'),0)
}


function foo() {
  console.log('foo')
  Promise.resolve().then(
    (str) =>console.log('micro-foo')
  )
  setTimeout((str) =>console.log('macro-foo'),0)

  bar()
}
foo()
console.log('global')
Promise.resolve().then(
  (str) =>console.log('micro-global')
)
setTimeout((str) =>console.log('macro-global'),0)
```

在这段代码中，包含了通过 setTimeout 宏任务和通过 Promise.resolve 创建的微任务，你认为最终打印出来的顺序是什么？

执行这段代码，我们发现最终打印出来的顺序是：

```
foo
bar
global
micro-foo
micro-bar
micro-global
macro-foo
macro-bar
macro-global
```

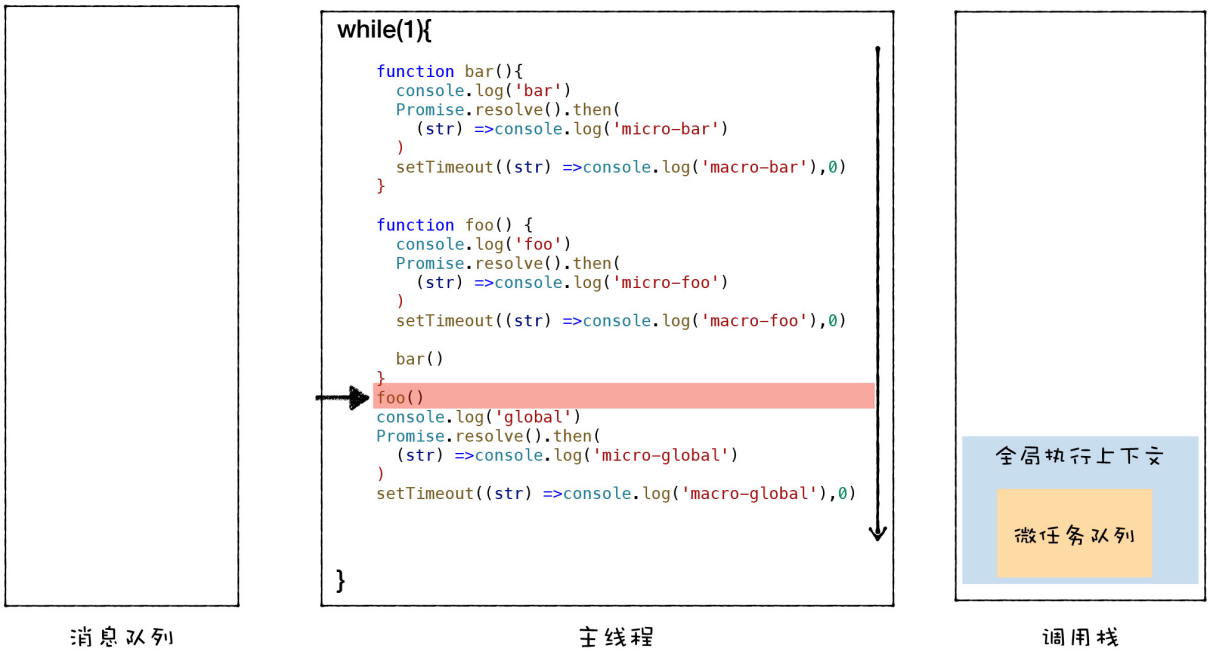
我们可以清晰地看出，微任务是处于宏任务之前执行的。接下来，我们就来详细分析下 V8 是怎么执行这段 JavaScript 代码的。

首先，当 V8 执行这段代码时，会将全局执行上下文压入调用栈中，并在执行上下文中创建一个空的微任务队列。那么此时：

调用栈中包含了全局执行上下文；

微任务队列为空。

此时的消息队列、主线程、调用栈的状态图如下所示：



然后，执行 foo 函数的调用，V8 会先创建 foo 函数的执行上下文，并将其压入到栈中。接着执行 Promise.resolve，这会触发一个 micro-foo1 微任务，V8 会将该微任务添加进微

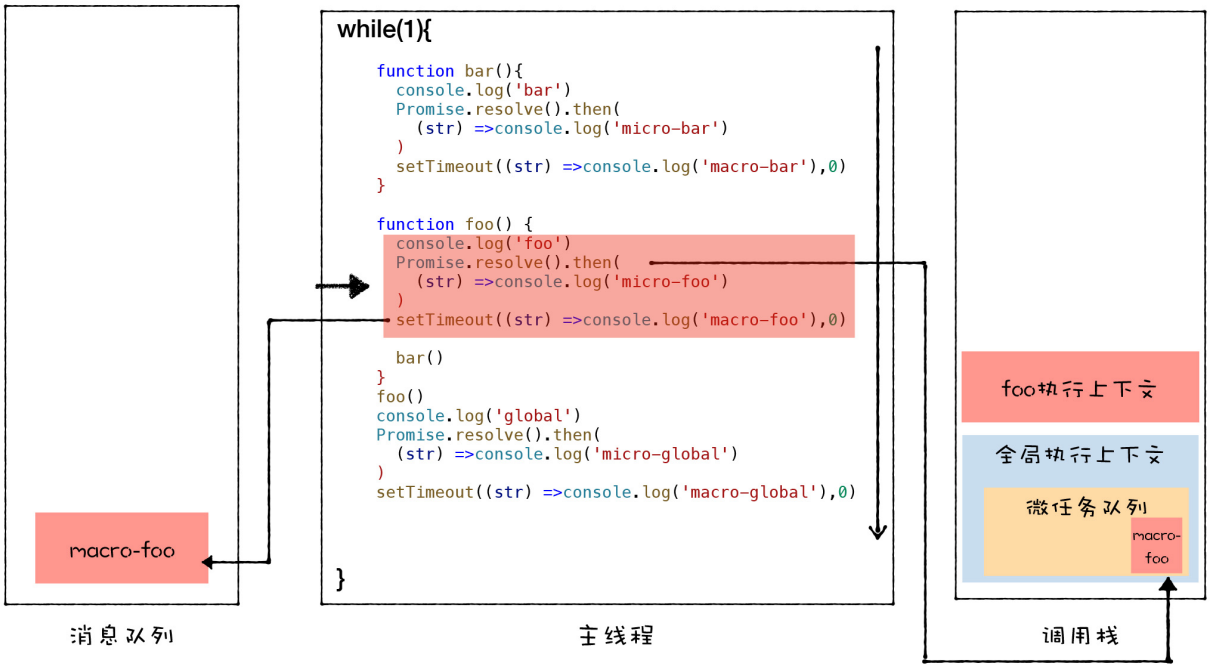
任务队列。然后执行 setTimeout 方法。该方法会触发了一个 macro-foo1 宏任务，V8 会将该宏任务添加进消息队列。那么此时：

调用栈中包含了**全局执行上下文、foo 函数的执行上下文**；

微任务队列有了一个微任务，**micro-foo**；

消息队列中存放了一个通过 setTimeout 设置的宏任务，**macro-foo。**

此时的消息队列、主线程和调用栈的状态图如下所示：



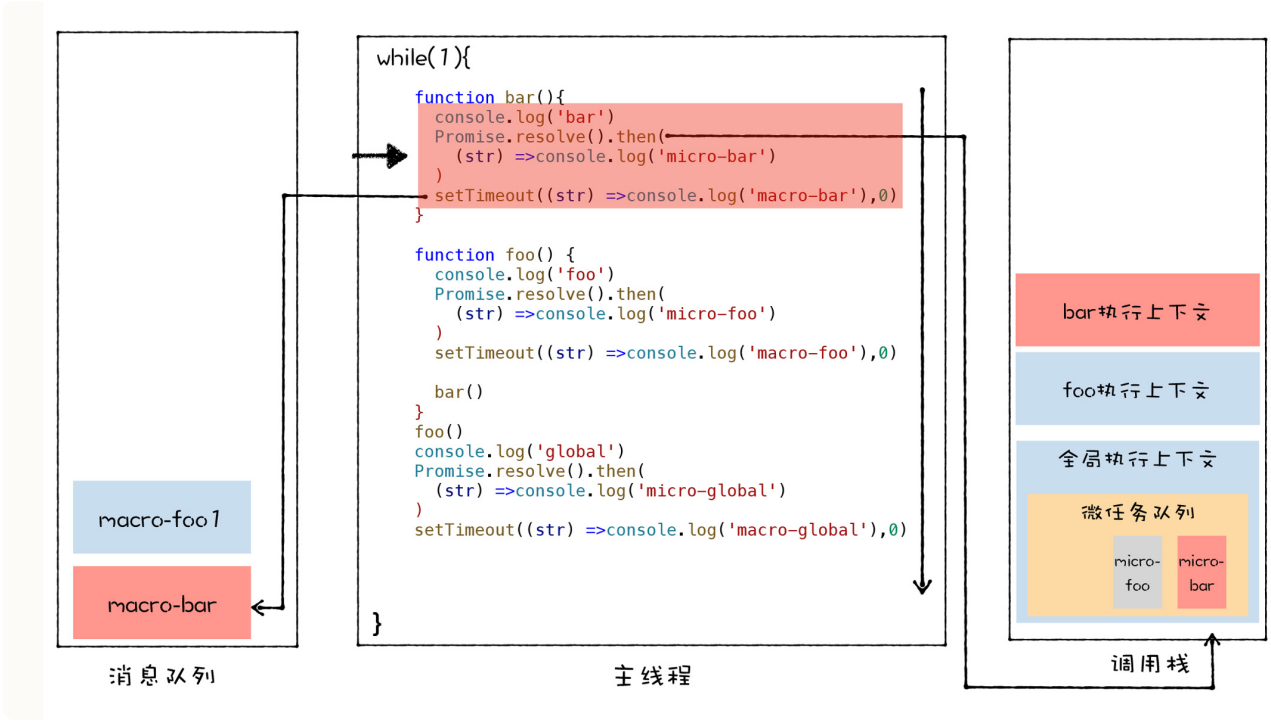
接下来，foo 函数调用了 bar 函数，那么 V8 需要再创建 bar 函数的执行上下文，并将其压入栈中，接着执行 Promise.resolve，这会触发一个 micro-bar 微任务，该微任务会被添加进微任务队列。然后执行 setTimeout 方法，这也会触发一个 macro-bar 宏任务，宏任务同样也会被添加进消息队列。那么此时：

调用栈中包含了**全局执行上下文、foo 函数的执行上下文、bar 的执行上下文**；

微任务队列中的微任务是 **micro-foo、micro-bar**；

消息队列中，宏任务的状态是 **macro-foo、macro-bar。**

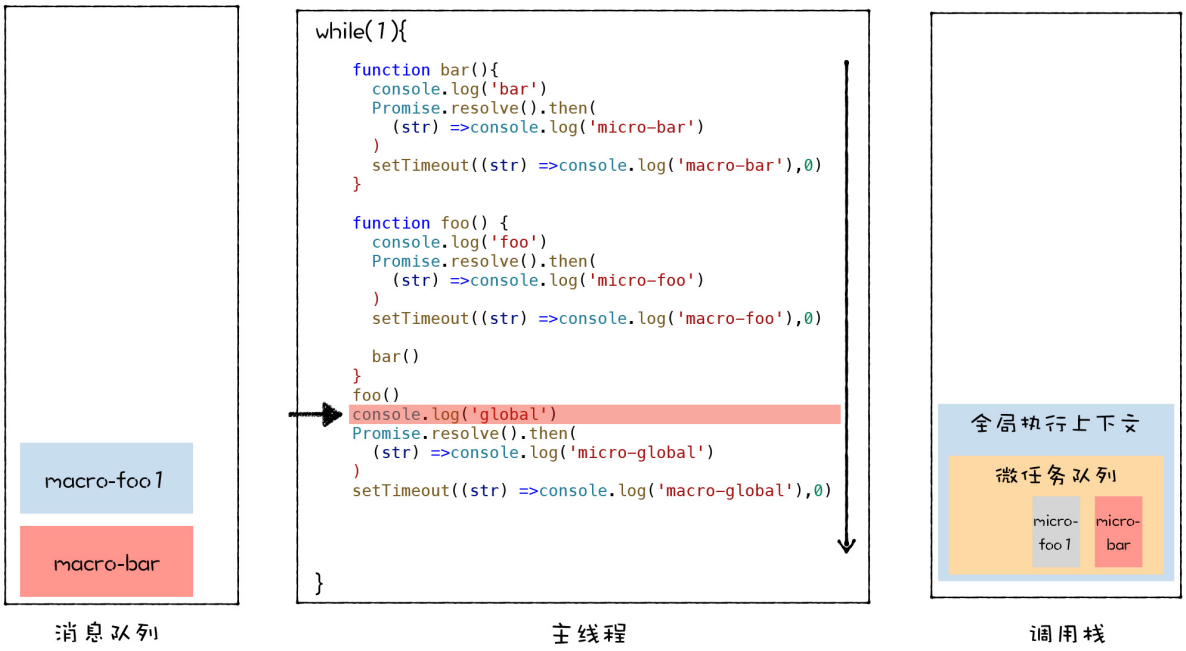
此时的消息队列、主线程和调用栈的状态图如下所示：



接下来，bar 函数执行结束并退出，bar 函数的执行上下文也会从栈中弹出，紧接着 foo 函数执行结束并退出，foo 函数的执行上下文也随之从栈中被弹出。那么此时：

> 调用栈中包含了**全局执行上下文**，因为 bar 函数和 foo 函数都执行结束了，所以它们的执行上下文都被弹出调用栈了；
>
> 微任务队列中的微任务同样还是 **micro-foo、micro-bar**；
>
> 消息队列中宏任务的状态同样还是 **macro-foo、macro-bar。**

此时的消息队列、主线程和调用栈的状态图如下所示：



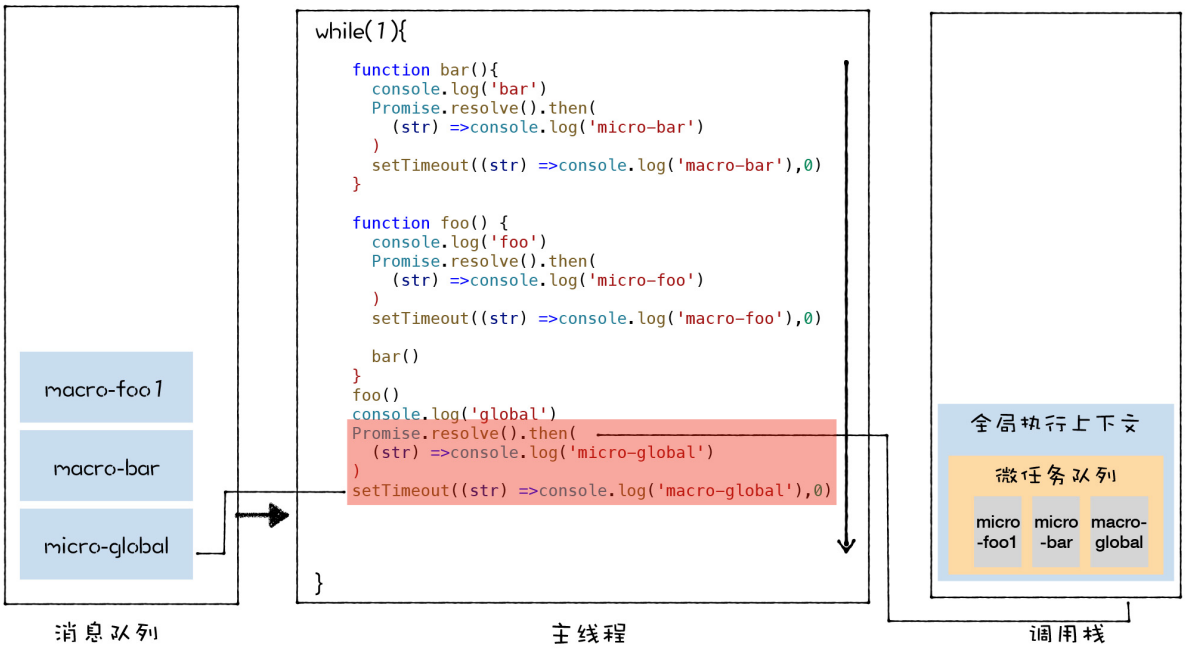
主线程执行完了 foo 函数，紧接着就要执行全局环境中的代码 Promise.resolve 了，这会触发一个 micro-global 微任务，V8 会将该微任务添加进微任务队列。接着又执行 setTimeout 方法，该方法会触发了一个 macro-global 宏任务，V8 会将该宏任务添加进消息队列。那么此时：

调用栈中包含的是**全局执行上下文**；

微任务队列中的微任务同样还是 **micro-foo、micro-bar、micro-global**；

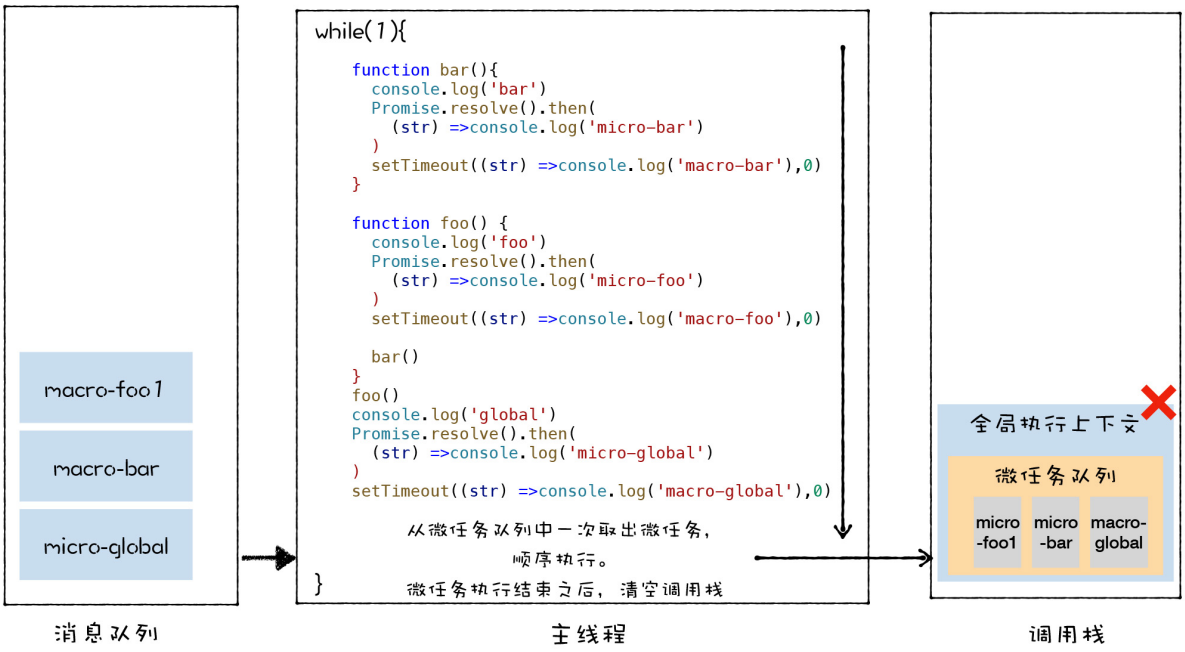
消息队列中宏任务的状态同样还是 **macro-foo、macro-bar、macro-global。**

此时的消息队列、主线程和调用栈的状态图如下所示：



等到这段代码即将执行完成时，V8 便要销毁这段代码的环境对象，此时环境对象的析构函数被调用（注意，这里的析构函数是 C++ 中的概念），这里就是 V8 执行微任务的一个检查点，这时候 V8 会检查微任务队列，如果微任务队列中存在微任务，那么 V8 会依次取出微任务，并按照顺行执行。因为微任务队列中的任务分别是：micro-foo、micro-bar、micro-global，所以执行的顺序也是如此。

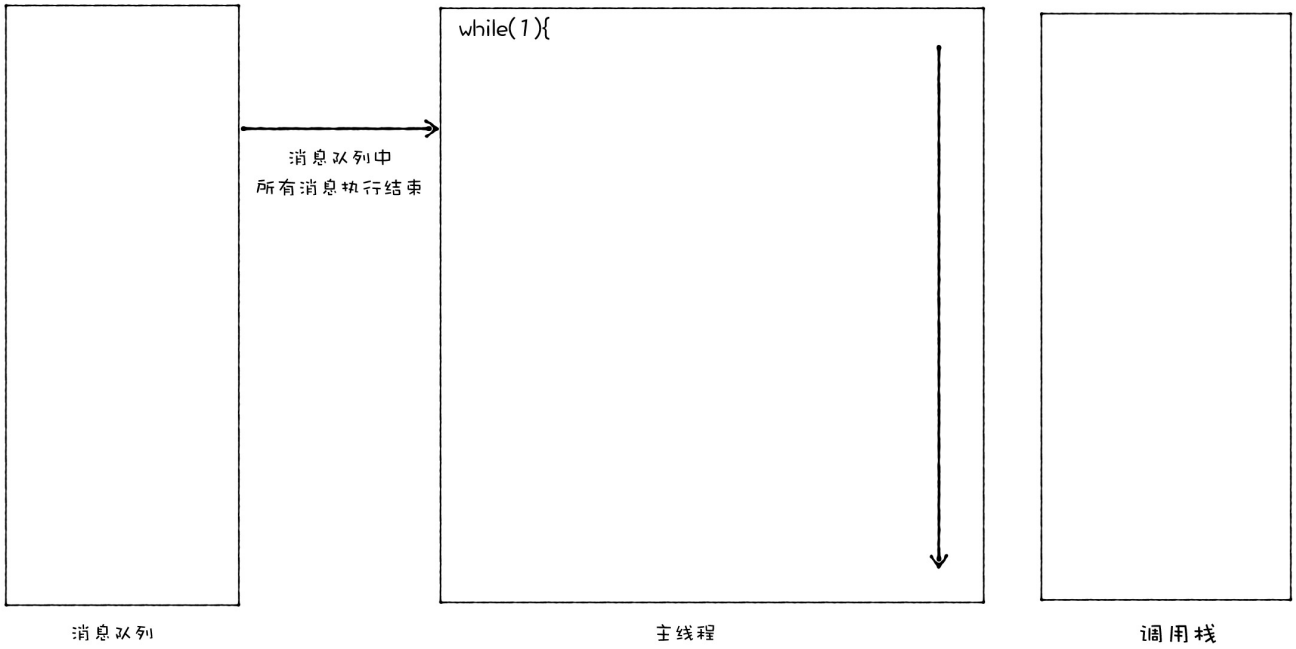
此时的消息队列、主线程和调用栈的状态图如下所示：



等微任务队列中的所有微任务都执行完成之后，当前的宏任务也就执行结束了，接下来主线程会继续重复执行取出任务、执行任务的过程。由于正常情况下，取出宏任务的循序是按照

先进先出的顺序，所有最后打印出来的顺序是：macro-foo、macro-bar、macro-global。

等所有的任务执行完成之后，消息队列、主线程和调用栈的状态图如下所示：



以上就是完整的执行流程的分析，到这里，相信你已经了解微任务和宏任务的执行时机是不同的了，微任务是在当前的任务快要执行结束之前执行的，宏任务是消息队列中的任务，主线程执行完一个宏任务之后，便会接着从消息队列中取出下一个宏任务并执行。

能否在微任务中循环地触发新的微任务？

既然宏任务和微任务都是异步调用，只是执行的时机不同，那能不能在 setTimeout 解决栈溢出的问题时，把触发宏任务改成是触发微任务呢？

比如，我们将代码改为：

复制代码

```
function foo() {
  return Promise.resolve().then(foo)
}
foo()
```

当执行 foo 函数时，由于 foo 函数中调用了 Promise.resolve()，这会触发一个微任务，那么此时，V8 会将该微任务添加进微任务队列中，退出当前 foo 函数的执行。

然后，V8 在准备退出当前的宏任务之前，会检查微任务队列，发现微任务队列中有一个微任务，于是先执行微任务。由于这个微任务就是调用 foo 函数本身，所以在执行微任务的过程中，需要继续调用 foo 函数，在执行 foo 函数的过程中，又会触发了同样的微任务。

那么这个循环就会一直持续下去，当前的宏任务无法退出，也就意味着消息队列中其他的宏任务是无法被执行的，比如通过鼠标、键盘所产生的事件。这些事件会一直保存在消息队列中，页面无法响应这些事件，具体的体现就是页面的卡死。

不过，由于 V8 每次执行微任务时，都会退出当前 foo 函数的调用栈，所以这段代码是不会造成栈溢出的。

总结

这节课我们主要从**调用栈**、**主线程**、**消息队列**这三者关联的角度来分析了微任务。

调用栈是一种数据结构，用来管理在主线程上执行的函数的调用关系。主线在执行任务的过程中，如果函数的调用层次过深，可能造成栈溢出的错误，我们可以使用 setTimeout 来解决栈溢出的问题。

setTimeout 的本质是将同步函数调用改成异步函数调用，这里的异步调用是将回调函数封装成宏任务，并将其添加进**消息队列**中，然后主线程再按照一定规则循环地从消息队列中读取下一个宏任务。

消息队列中事件又被称为宏任务，不过，宏任务的时间颗粒度太粗了，无法胜任一些对精度和实时性要求较高的场景，而**微任务可以在实时性和效率之间做有效的权衡**。

微任务之所以能实现这样的效果，主要取决于微任务的执行时机，**微任务其实是一个需要异步执行的函数，执行时机是在主函数执行结束之后、当前宏任务结束之前。**

因为微任务依然是在当前的任务中执行的，所以如果在微任务中循环触发新的微任务，那么将导致消息队列中的其他任务没有机会被执行。

思考题

浏览器中的 MutationObserver 接口提供了监视对 DOM 树所做更改的能力，它在内部也使用了微任务的技术，那么今天留给你的作业是，查找 MutationObserver 相关资料，分析它是如何工作的，其中微任务的作用是什么？欢迎你在留言区与我分享讨论。

感谢你的阅读，如果你觉得这一讲的内容对你有所启发，也欢迎把它分享给你的朋友。

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇　17 | 消息队列：V8是怎么实现回调函数的?

下一篇　19 | 异步编程（二）：V8是如何实现async/await的?

精选留言 (11)

写留言

William

2020-04-26

请问老师为何把微任务队列画在全局执行上下文内，有什么依据吗?

1　2

成楠Peter
2020-04-26

思考题，MutationObserver和IntersectionObserver两个性质应该差不多。我这里简称ob。ob是一个微任务，通过浏览器的requestIdleCallback，在浏览器每一帧的空闲时间执行ob监听的回调，该监听是不影响主线程的，但是回调会阻塞主线程。当然有一个限制，如果100ms内主线程一直处于未空闲状态，那会强制触发ob。

展开

1

零和幺
2020-04-26

像 setTimeout 、XMLHttpRequest 这种 web APIs，是浏览器的哪个部分提供的呢？它们并不是 V8 提供的，是浏览器内核么？提供这些 web APIs 的部分与 V8 又有什么关系？

1

伏枫
2020-04-26

老师，哪些是宏任务，哪些是微任务？

展开

1

sugar
2020-04-25

老师您好，我按照咱们的课程介绍把v8项目的开发环境部署好了，通过gclient同步代码、通过ninja编译构建d8可执行文件。但是遇到一个问题：我想在v8源代码中自己加断点看各个环节的运行情况，在c++里不能像js那样愉快地console.log任何一个对象出来，于是ninja编译d8时靠std::cout或者printf无法输出代码中的任何一个类对象；而如果能用xcode调试这些c代码，也可以借助其breakpoint来解决，但我通过gn gen out/gn --ide=xcode...

展开

3

1

code-artist
2020-04-28

微任务执行时，还是会在调用栈中创建对应函数的执行上下文吗？

作者回复: 会的，和正常执行函数一样

leaf

2020-04-27

老师写的很精彩，看起来一点不累

展开

伟龙

2020-04-26

setTimeout应该是由专门的定时器线程去管理吧，到点了才插入消息队列，然后等待消费

luckyone

2020-04-26

当前环境微任务执行不完怎么处理，或者有这种情况吗，第二个是dom事件是宏任务？

1

tt

2020-04-25

每次宏任务结束都会清空调用栈，那如何在宏任务之间共享信息呢？

洋洋

2020-04-25

老师，对于微任务的执行时机"在当前宏任务结束之前"不是很理解。对于文中的例子，我可以理解为在执行微任务时当前的宏任务就是主线程最后一行的setTimeout函数调用吗？